

JS Level 1



Введение



ОСНОВЫ JS

В этой лекции мы с вами поговорим об основах JS – переменных, типах данных, ключевых операторах и приведении типов.

Ключевое: мы вас будем с самого начала приучать пользоваться инструментами. Это вам поможет затем сэкономить огромное количество времени при написании кода и его отладке (приведении в рабочее состояние).

Поэтому не ленитесь и самостоятельно повторяйте все те действия, что описаны в лекции, т.к. если вы ничего не делаете руками, а просто листаете презентацию, то вы ничему не научитесь, а значит зря тратите своё время.



ОСНОВЫ JS

Ключевой темой сегодняшней лекции будет работа с числами и переменными.

Мы не стремимся вас "накачать" сразу всеми тонкостями языка (заставить вас выучить все WAT – можете погуглить, что это такое), вместо этого мы учим вас писать код правильно и нарабатывать практические навыки.



Повторение



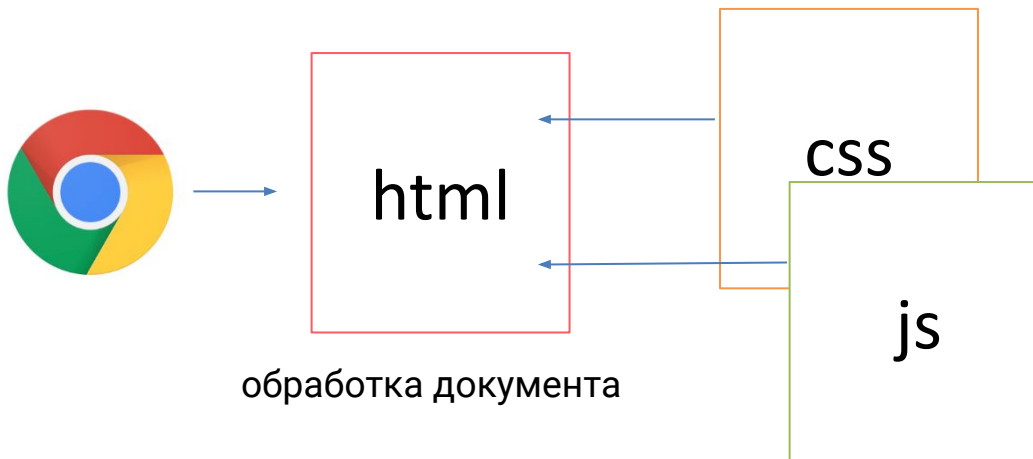
Web Application

На прошлой лекции мы поговорили с вами, как работают веб-приложения (а именно их клиентская часть) – они загружаются и запускаются в браузере:



Ресурсы

Кроме того, мы обсудили сам механизм: сначала загружается HTML-документ (если вы указали его в адресной строке), а затем уже все ресурсы, которые в этом самом документе прописаны:



JS

В первую очередь, нас интересует именно JS* – возможность писать приложения (или программы), которые будут выполняться в браузере. И мы сразу с вами будем рассматривать реальные задачи, чтобы научиться их решать.

Примечание*: но про CSS и HTML вы также не должны забывать. Они (CSS и HTML) сейчас умеют достаточно многое и без привлечения JS.



JS

Одна из задач - виджет перевода денег. Виджет – это такой небольшой самостоятельный компонент на веб-странице, который умеет делать хорошо одну задачу. Так вот наш виджет выглядит следующим образом:

Валюта	Сумма	от 5 до 4000 TJS
TJS	<input type="text" value="10000"/>	<input type="button" value="X"/>
Курс конвертации	1 ₸ = 0,147 сомони	
Сумма к зачислению	68 027,21 ₸	
Комиссия alif mobi	1.0%	
Итого к оплате	10 100,00 сомони	
<input type="button" value="ДАЛЕЕ"/>		



JS

Ключевая идея: пользователь вводит сумму в поле "Сумма", автоматически высчитываются все значения, и, если они пользователю нравятся, то он нажимает на кнопку "Далее" и переводит средства.

Валюта	Сумма	от 5 до 4000 TJS
TJS	10000	X
Курс конвертации	1 ₸ = 0,147 сомони	
Сумма к зачислению	68 027,21 ₸	
Комиссия alif mobi	1.0%	
Итого к оплате	10 100,00 сомони	

ДАЛЕЕ



JS

Первое, чему мы должны научиться - это декомпозиция задач (разбиваем задачи на мелкие подзадачи). Несмотря на то, что виджет всего один, сразу можно выделить целых три подзадачи:

1. Визуальная составляющая: как пользователь будет вводить данные, как вы будете выводить ему результат (включая вёрстку, стили)
2. Алгоритм: формулы, расчёты (деньги нужно ещё конвертировать)
3. Взаимодействие с сервером (деньги пользователя не хранятся в браузере, они хранятся на счету в банке)

→ можем сделать уже сейчас



Идеальные условия

Научитесь не ждать "идеальных условий". Т.к. их никогда не будет. Но если вы сможете сделать часть задачи уже сейчас, то не придётся делать "всё сразу" в последний момент.



Проект

Создадим проект, который будет называться `transfer` и, как обычно, три файла:

1. `index.html`
2. `css/styles.css`
3. `js/app.js`

Надеемся, вы помните, как их подключать (если нет, то пересмотрите предыдущую лекцию).



Проект

На всякий случай напоминаем, как должен выглядеть ваш [index.html](#) (не забывайте про Emmet):

```
<> index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Document</title>
7    <link rel="stylesheet" href="css/styles.css">
8  </head>
9  <body>
10   <script src="js/app.js"></script>
11 </body>
12 </html>
```



Формулы

В принципе, всё достаточно просто: нам (как в школе) просто нужны формулы, по которым мы будем считать.

В реальных проектах вам могут давать готовые формулы и примеры расчёта по этим формулам (это идеальный вариант). Но бывает и так, что вам придётся самим эту формулы "вывести".

В нашем случае формулы расчёта будут такие:

- для суммы к зачислению: $10\,000 / 0.147 = 68\,027.21$
- для суммы платежа: $10\,000 * 1.01 = 10\,100$



Языки программирования

До этого мы с вами обсуждали HTML и говорили, что это язык, предназначенный для описания документов. В нём есть специальные элементы, со своими атрибутами, правилами вложенности и т.д.

Но что же такое язык программирования? Язык программирования – это специальный язык, который представляет нам возможность заставить производить компьютер какие-то действия, например, считать (что нам и нужно).



Формула

Если мы говорим про "считать", то первая аналогия - это калькулятор.

Давайте попробуем записать всё как в калькуляторе и попробовать запустить:

```
JS app.js  ×   
js > JS app.js  
1 10000 / 0.147
```

Обратите внимание, в числе пробелы не пишутся, и в качестве разделителя дробной и целой части выступает . (точка).

Запускаем Live Server, открываем нашу страницу и... ничего не происходит (или мы просто не видим, что что-то происходит).

Давайте разбираться.



Формула

В браузер встроен "[движок](#)" ([engine](#)) JS, который занимается тем, что разбирает те инструкции, которые мы написали и выполняет их (это как переводчик – например, вы пытаетесь прочитать веб-страницу на незнакомом вам языке, сами вы этого сделать не можете, но можете воспользоваться [Google Translate](#) – он вам и переведёт и озвучит).

Text input fields for translation, showing "АНГЛИЙСКИЙ" and "РУССКИЙ" selected. The interface includes a microphone icon and a character count "0 / 5000".

[Отправить отзыв](#)



Формула

Для простоты пока будем считать, что как только браузер обнаруживает JS-файл (подключенный через тег `script`), он подключает этот движок для разбора файла и выполнения инструкций в нём.

На данный момент нам важно увидеть, что это действительно происходит.



Debugger



Debugger

Для того, чтобы увидеть, как движок JS исполняет наш файл (и исполняет ли вообще), в Developer Tools встроен специальный инструмент, который называется **Debugger** (отладчик).

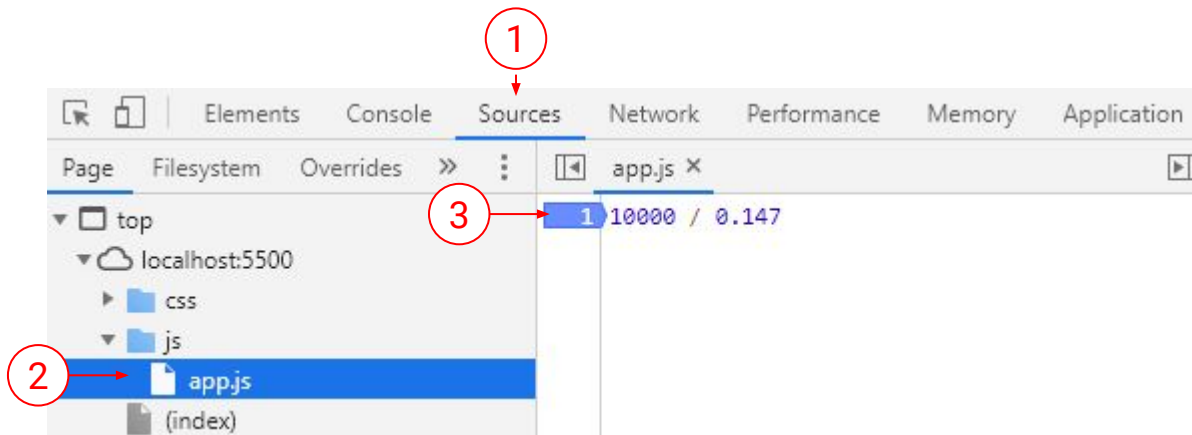
Отладчик – это специальный инструмент, который позволяет перевести движок JS в режим пошагового выполнения. При этом мы можем смотреть, что и как выполняется.

Далее для краткости мы не будем говорить "движок JS", а будем просто говорить браузер.

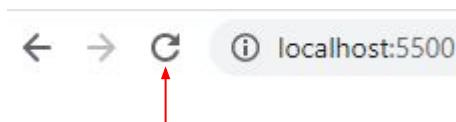


Debugger

Заходим в DevTools, открываем вкладку **Sources** (1), в боковой панели выбираем файл **app.js** (2) и на поле с номерами строк кликаем левой кнопкой мыши один раз (чтобы установилась "точка остановки"):

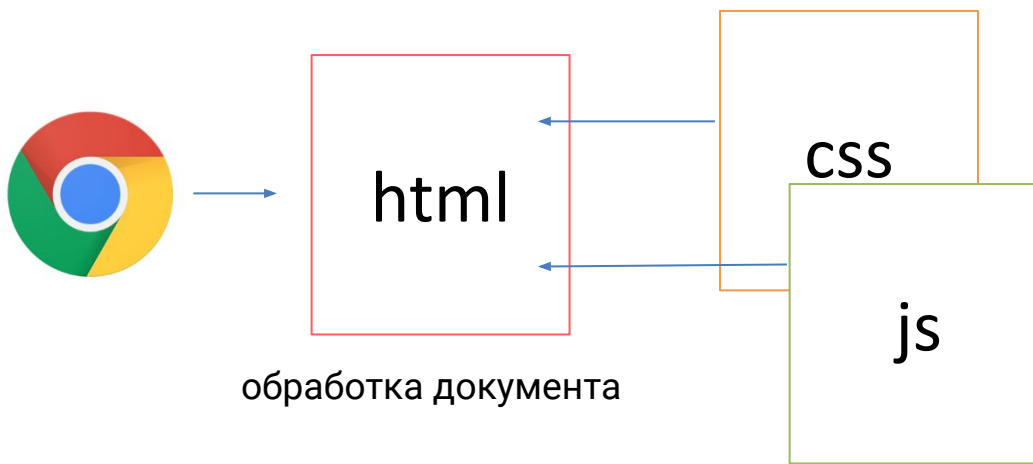


Точка остановки – это строка, на которой остановится выполнение. Для того, чтобы её активировать, нужно перезагрузить страницу (**F5**) или кнопка обновления страницы:



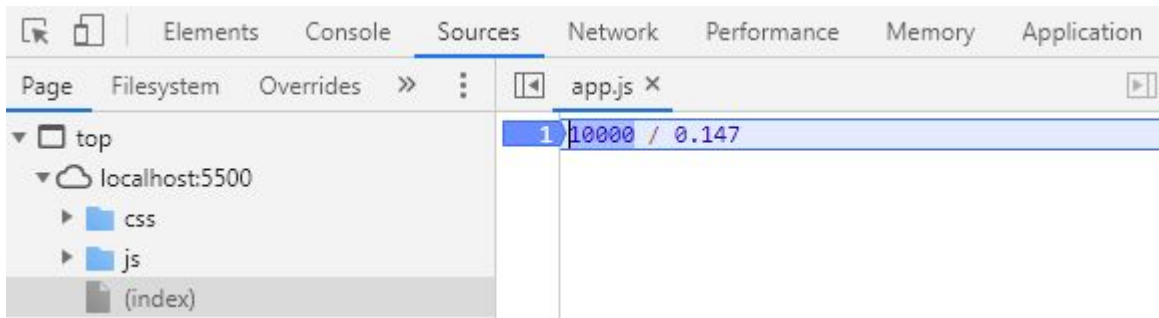
Debugger

Обратите внимание, браузер сейчас исполняет нашу программу только при перезагрузке, т.е. в этот момент он заново обрабатывает [index.html](#), подгружает [app.js](#) и выполняет:



Debugger

После обновления строка с формулой подсветится (это значит, что браузер ещё эту строку не выполнил):



Кроме того, на самой странице (и в панели справа появятся кнопки управления):



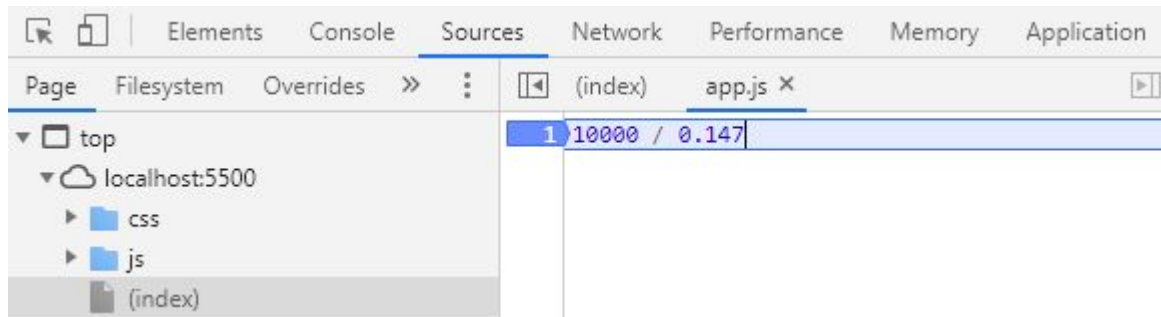
Нас будут интересовать только две:

- ▶ продолжить дальше, пока не встретится следующая точка остановки
- ↻ сделать один "шаг" вперёд

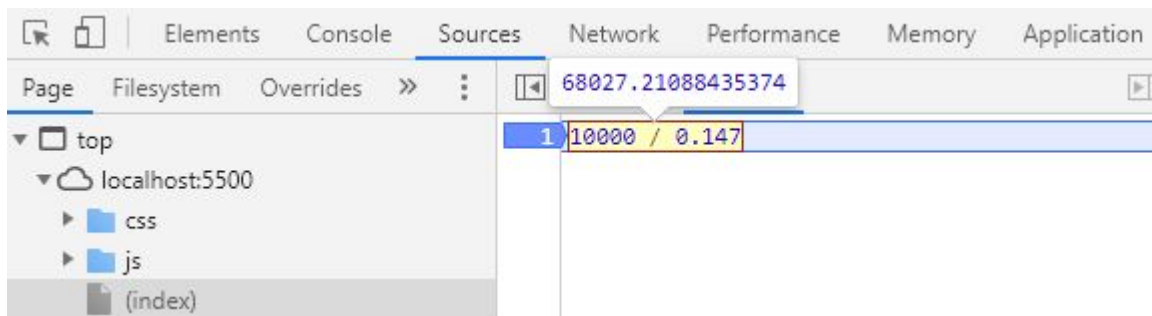


Debugger


Нажмите на шаг вперёд и увидите, что браузер установит курсор в конец нашей "формулы":

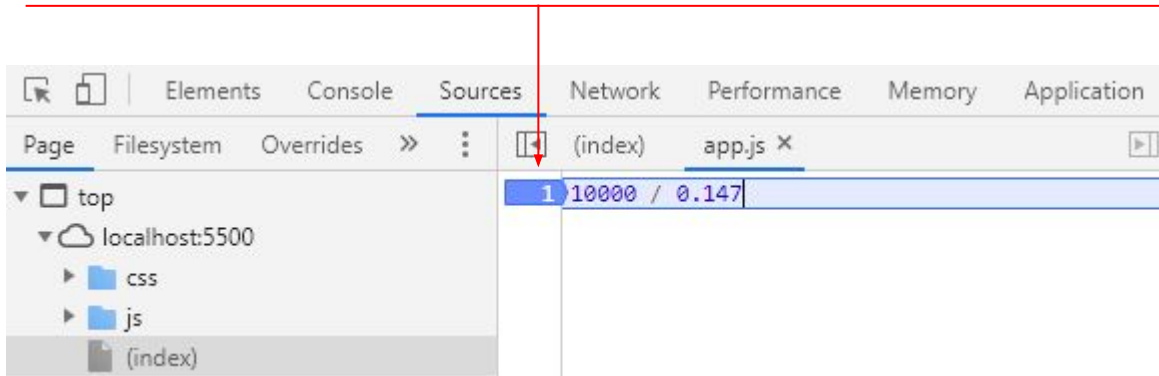


Это значит, он вычислил результат нашего выражения. Вы можете попросить это сделать его для вас ещё раз и посмотреть на результат, выделив всё выражение (как обычный текст):



Debugger

Если вы нажмёте ещё раз на шаг вперёд, то попадёте в [index.html](#). Поэтому просто уберите точку остановки (кликните ещё раз на боковое поле) и нажмите на кнопку 



Теперь браузер не будет после каждого обновления страницы останавливаться в данной точке.



Debugger

Мы с вами посмотрели, что браузер действительно что-то делает с нашим выражением, но куда девается результат?

На самом деле, всё достаточно просто - поскольку мы ничего не делаем с результатом, браузер просто про него "забывает" (это, конечно, упрощённое описание, но нам подходит).



Debugger

На прошлой лекции мы уже использовали конструкцию `alert` для вывода текста во всплывающем окне. Давайте попробуем повторить этот же трюк:

JS app.js ×

js > JS app.js

```
1 alert(10000 / 0.147);
```



Подтвердите действие на странице localhost:5500

68027.21088435374

OK



Как это работает?

Так же, как в математике, в JS есть понятие функций и приоритета. Например, если вы вспомните математику, то там были выражения вроде $\sin(x)$. Если мы перенесём на наш пример, то `alert(10000/0.147)` аналогично (по смыслу), например, $\sin(3.14 / 2)$.

Так же, как в математике, сначала вычисляет то, что в скобках, а потом вызывается функция с результатом:

1. Шаг 1: $10000/0.147 = 68027.21088435374$
2. Шаг 2: `alert(68027.21088435374)`



alert, prompt, confirm

Несмотря на то, что этот способ работает – он считается крайне дурным тоном, кроме того обладает побочными эффектами, например "замораживает" исполнение любого другого кода в открытой вкладке браузера. Поэтому если ваш код в домашних заданиях будет содержать [alert](#) (как и [confirm](#), [prompt](#)), то мы будем отправлять его на доработку.

Современный подход подразумевает использование элементов на веб-странице для ввода и вывода данных. Но это будет в следующих лекциях (для этого нам нужно достаточно много чего изучить), поэтому мы воспользуемся DevTools и вызовом [console.log](#) для печати сообщения в консоль DevTools.



console.log



console.log

Неотъемлемой частью DevTools является консоль ([Console](#)). Туда выводятся сообщения об ошибках, вспомогательные сообщения и даже предупреждения для пользователей:

```
.d8888b. 888      888
d88P  Y88b 888      888
Y88b.      888      888
"Y888b.    888888 .d88b. 88888b. 888
    "Y88b. 888    d88""88b 888 "88b 888
      "888 888    888 888 888 888 Y8P
Y88b d88P Y88b. Y88..88P 888 d88P
"Y8888P"  "Y888 "Y88P" 88888P" 888
                        888
                        888
                        888
```

This is a browser feature intended for developers. If someone told you to copy and paste something here to enable a Facebook feature or "hack" someone's account, it is a scam and will give them access to your Facebook account.

See <https://www.facebook.com/selfxss> for more information.



console.log

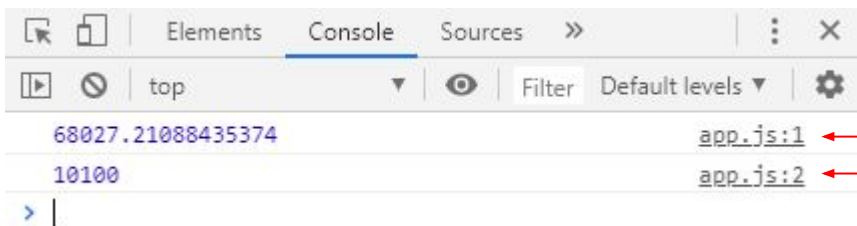
Почему в названии присутствует точка (`console.log`) мы с вами поговорим чуть позже, пока же для нас важно, что этот вызов позволяет выводить значения в консоль браузера:

```
JS app.js × 
```

```
js > JS app.js  
  1  console.log(10000 / 0.147);  
  2  console.log(10000 * 1.01);
```

← в конце строки ставится ;

Сама консоль расположена на вкладке **Console** в Developer Tools:



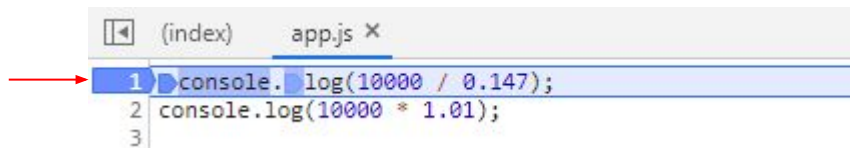
номера строк,
на которых производился вывод

Консоль браузера (не `console.log`) – достаточно мощный инструмент, им обязательно нужно научиться пользоваться.



console.log

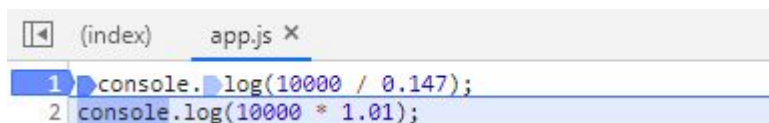
Вернёмся к дебаггеру и снова поставим точку остановки на первой строке:



Если мы сейчас переключимся на консоль, то там будет "пусто", поскольку наша строка ещё не отработала:



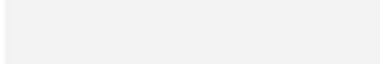
Но если мы сделаем шаг вперёд, то информация в консоли появится:



console.log

Мы с вами посмотрели как пользоваться отладчиком. На последнем примере увидели, что браузер исполняет наш код по строкам сверху-вниз и немного поговорили о том, как всё работает.

Но ключевая проблема теперь в следующем: представьте, что вы через месяц открываете этот код и пытаетесь понять, что значит каждое из чисел:

```
JS app.js ×   
js > JS app.js  
1 console.log(10000 / 0.147);  
2 console.log(10000 * 1.01);
```

Возможно, вы даже вспомните, но на это уйдёт время.



Переменные



Переменные

Чтобы не приходилось вспоминать, в JS есть возможность объявлять переменные.

Переменные – это просто удобные имена для хранения и использования информации.

Это работает так же, как и с людьми: мы даём людям имена (и используем их), чтобы нам было удобнее к ним (к людям) обращаться, запоминать и т.д.



Переменные

Переменные объявляются с помощью ключевых слов `let` или `const`* и выглядит это следующим образом:

JS app.js ×

js > JS app.js > ...

```
1  const input = 10000;
```

← привязали к именам значения

```
2  const exchangeRate = 0.147;
```

```
3  const commission = 0.01;
```

```
4
```

```
5  const amount = input / exchangeRate;
```

```
6  const payment = input * (1 + commission);
```

теперь можем использовать имя вместо того, чтобы везде писать значение

```
7
```

```
8  console.log(amount);
```

```
9  console.log(payment);
```

Примечание*: есть ещё `var`, но он используется только в случае необходимости поддержки старых браузеров/кода (при этом этот код чаще всего генерируется автоматически из нового кода).



Переменные

Давайте разбираться:

1. Ключевое слово `const` говорит, что мы связываем имя со значением всего один раз (это пока будет сложно понять, поэтому пока просто привыкните везде по возможности использовать `const`):

JS app.js X

js > JS app.js > ...

```
1  const input = 10000;  
2  const exchangeRate = 0.147;  
3  const commission = 0.01;
```

2. Оператор `=` занимается тем, что вычисляет то выражение, которое написано справа, а затем привязывает его к имени, расположенному слева:

```
5  const amount = input / exchangeRate;  
6  const payment = input * (1 + commission);
```

При этом вместо имён переменных (например, `input`) подставляются те значения, которые к ним "привязаны" (т.е. `10000`).



Переменные

Давайте разбираться:

1. Ключевое слово `const` говорит, что мы связываем имя со значением всего один раз (это пока будет сложно понять, поэтому пока просто привыкните везде по возможности использовать `const`):

JS app.js X

js > JS app.js > ...

```
1  const input = 10000;  
2  const exchangeRate = 0.147;  
3  const commission = 0.01;
```

2. Оператор `=` занимается тем, что вычисляет то выражение, которое написано справа, а затем привязывает его к имени, расположенному слева:

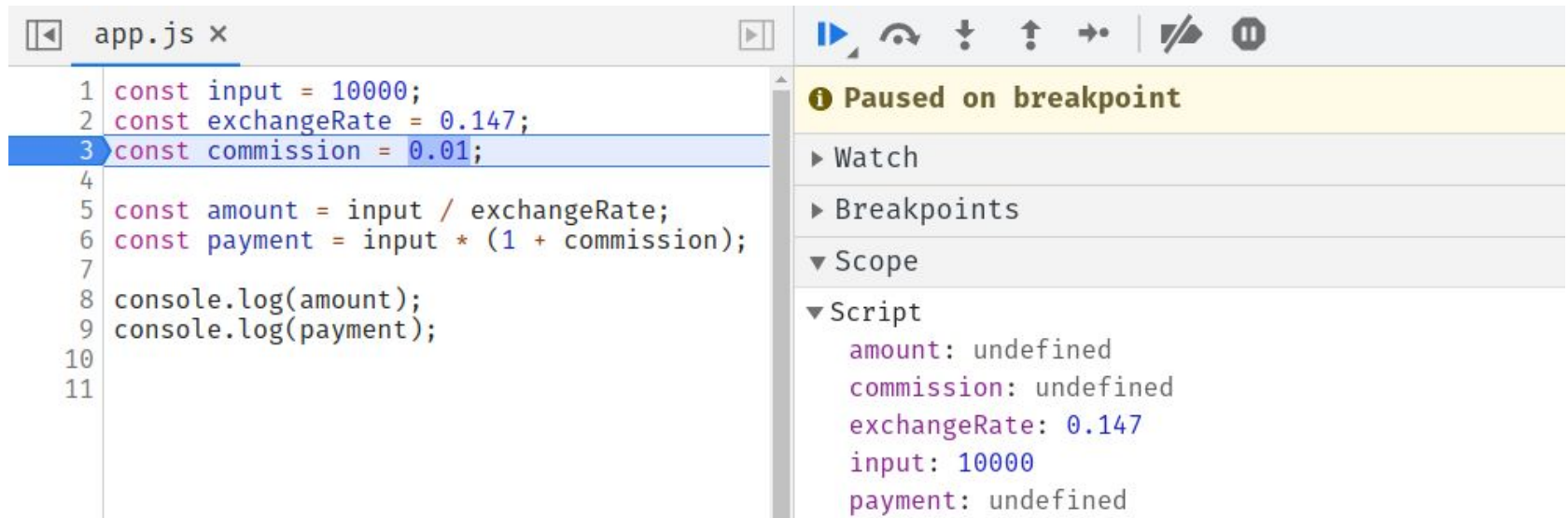
```
5  const amount = input / exchangeRate;  
6  const payment = input * (1 + commission);
```

При этом вместо имён переменных (например, `input`) подставляются те значения, которые к ним "привязаны" (т.е. `10000`).

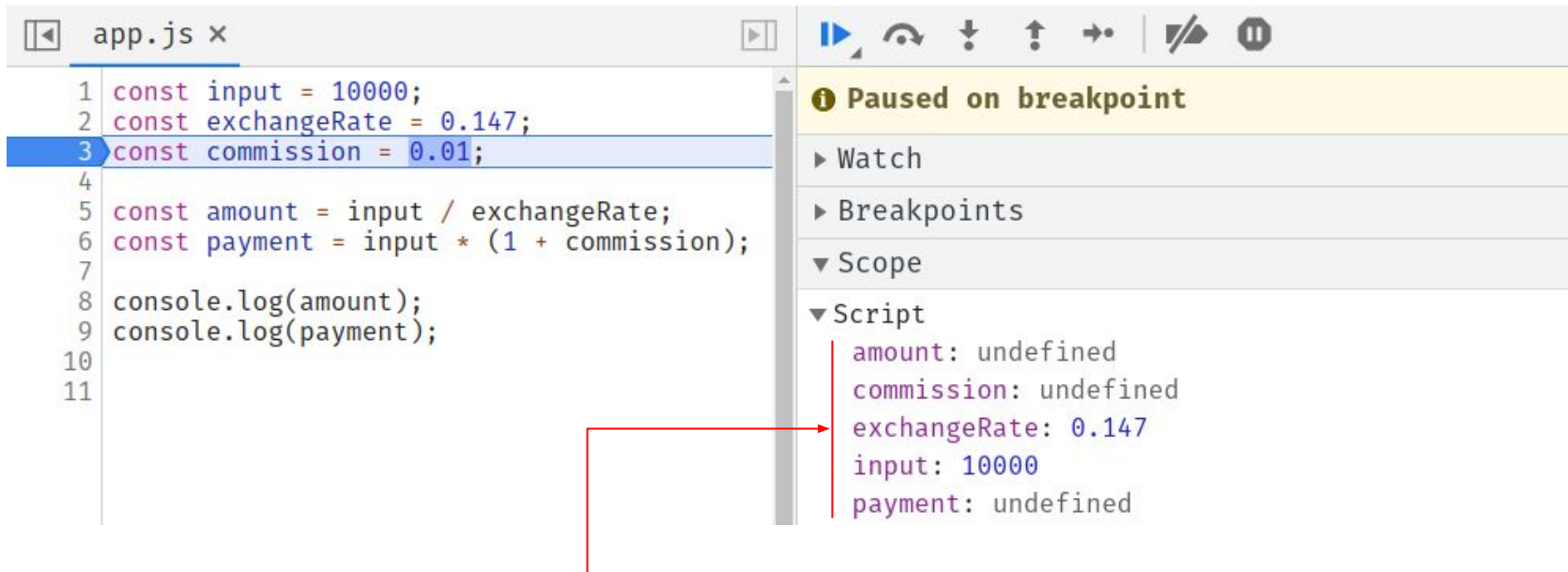


Переменные

Давайте посмотрим, как это всё работает в отладчике. Сейчас мы находимся на 3-ей строке (она подсвечена, а значит браузер её ещё не выполнил):



Переменные



В боковой панели, в разделе **Scope** будут "заполняться" имена по мере того, как вы будете "прошагивать" строки. Т.е. имена "заполняются" тогда, когда выполнена строка, в которой содержится объявление.



Переменные

Важно: переменные нужны для двух вещей:

1. Давать имена данным (поэтому имена переменных должны быть понятны):
 - имена принято писать на английском, потому что сейчас большинство проектов – международные, и если каждый участник проекта будет писать на своём языке, то они друг друга просто не пойму
 - важно: **summa**, **procent** – это не английский язык!
2. Использовать имена вместо того, чтобы руками менять данные везде: чтобы сосчитать данные не для **10 000**, а для **20 000**, нам достаточно поменять значение только в одном месте

JS app.js

×

js > JS app.js > ...

1 const input = 20000;



Имена переменных

Имена переменных принято писать с маленькой буквы, каждое следующее слово начиная с большой: `exchangeRate`.

Обычно в качестве имён используются одно-три слова, не больше.

Общие правила, применимые к именам:

1. Должно быть осмысленным
2. Начинается с буквы (`_`, `$` – не рекомендуется)
3. Содержит буквы, цифры, (`_`, `$` – не рекомендуется)
4. Регистрозависимо (`exchangerate` и `exchangeRate` – это разные имена)

В компаниях, где строго следят за качеством кода, применяют специальные инструменты – линтеры, которые в автоматическом режиме следят за тем, что вы выполняете эти правила (конечно, первое правило они проверить не могут, но в остальном – вполне).



Имена переменных

С переменными есть только одна сложность – им трудно придумывать хорошие имена. Но надо стараться, потому что вот такой код никуда не годится*:

```
JS app.js  X
js > JS app.js > ...
1  const a = 10000;
2  const b = 0.147;
3  const c = 0.01;
4
5  const d = a / b;
6  const e = a * (1 + c);
7
8  console.log(d);
9  console.log(e);
```

Примечание*: если вы покажете такой код при устройстве на работу, скорее всего, с вами даже не будут разговаривать.



const vs let

Ключевое слово **const** позволяет всего один раз связать имя и значение (чаще говорят присвоить). Ключевое слово **let** позволяет "перепривязывать" к имени значения.



const vs let

В современном мире рекомендуется использовать `const`, а не `let`.

Q: почему, ведь `let` позволяет делать больше?

A: относитесь к `const` как к одноразовой посуде – её нельзя переиспользовать, если она уже кем-то использована. Это защищает нас от того, чтобы случайно, где-то в глубине программы, не присвоить имени другого значения.

Т.е. ключевая идея – использовать инструменты, которые защитят нас от случайных ошибок, на поиски которых мы можем потратить потом своё время.



const vs let

Если мы попробуем случайно это сделать (переприсвоить значение), то получим ошибку, которая "обрушит" всю нашу программу:

JS app.js ×

js > JS app.js > ...

```
1  const input = 10000;  
2  const exchangeRate = 0.147;  
3  const commission = 0.01;  
4  
5  const input = 20000;
```

✖ Uncaught SyntaxError: Identifier 'input' has already been declared

app.js:5

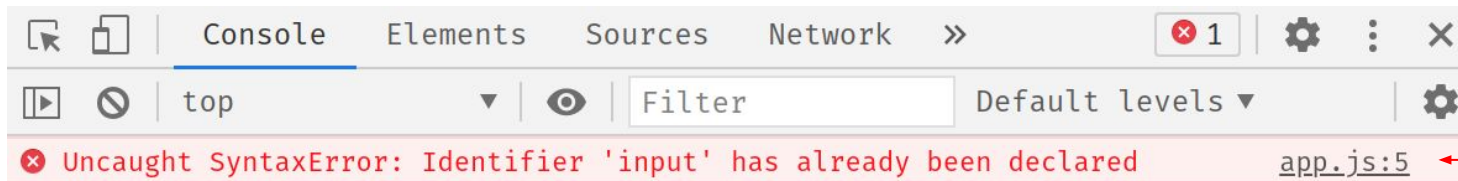
>

В консоль ничего не распечаталось (имеется в виду `amount` и `payment`), т.к. это – ошибка и выполнение программы было завершено в той точке, где произошла ошибка.



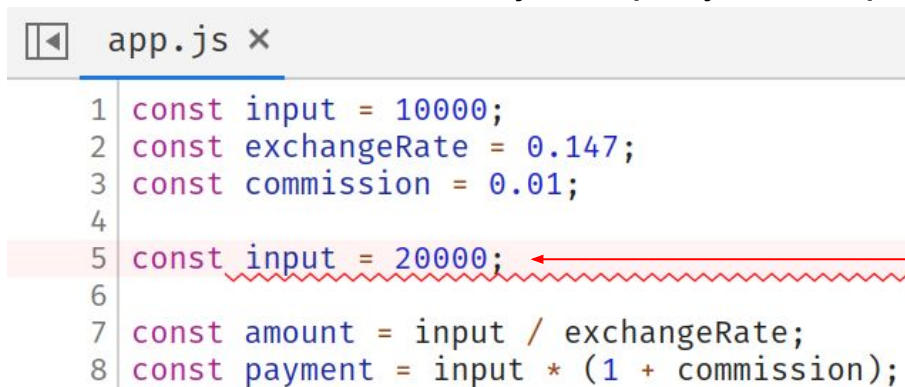
Ошибки

Важно: всегда смотрите на ошибки в консоли: если они там есть, значит в вашей программе что-то работает неправильно и нужно разобраться и исправить это:



Пример ошибки (ошибки помечаются специальной иконкой с крестиком ).

Вы всегда можете кликнуть строку, в которой произошла ошибка и попадёте в код:



Переменные

Хорошие практики:

1. Используйте по возможности всегда `const`
2. Не бойтесь создавать переменные (это не дорого)
3. Сначала кладите всё в переменную, а потом уже выводите в консоль:

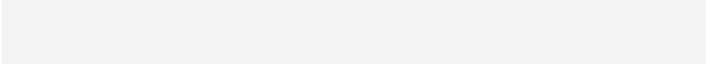
```
const amount = input / exchangeRate;  
const payment = input * (1 + commission);
```

```
console.log(amount);  
console.log(payment);
```



ИТОГОВЫЙ КОД

Итак, мы с вами поработали с переменными, оператором присваивания (=) и некоторыми арифметическими операторами. Итоговый код нашей программы для виджета пока выглядит вот так:

```
JS app.js ×   
js > JS app.js > ...  
1  const input = 10000;  
2  const exchangeRate = 0.147;  
3  const commission = 0.01;  
4  
5  const amount = input / exchangeRate;  
6  const payment = input * (1 + commission);  
7  
8  console.log(amount);  
9  console.log(payment);
```

Он ещё не имеет интерфейса, но расчёт производится.



Переменные

Ключевые моменты: переменная – это некоторое имя с привязанным к нему значением.

Значение какого типа хранится в переменной определяет то, какие операции с ней (переменной) можно выполнять.

- Имя
- Значение
- Тип* (определяется значением – поговорим чуть позже)

Давайте подробнее поговорим про операторы (те операции, которые можно выполнять с переменными).



Операторы



Операторы

Ключевыми для работы с числами являются арифметические операторы:

1. $a + b$ – оператор сложения, например, $10 + 3 = 13$
2. $a - b$ – оператор вычитания, например, $10 - 3 = 7$
3. a / b – оператор деления, например, $10 / 3 = 3.3333333333333335^*$
4. $a * b$ – оператор умножения, например, $10 * 3 = 30$
5. $a \% b$ – остаток от деления, например, $10 \% 3 = 1$
6. $a ** b$ – возведение в степень, например, $10 ** 3 = 1000$

Важно запомнить: набор операторов фиксирован. Мы не можем добавить свой или "переделать" существующий.

Примечание*: в компьютерах "компьютерная" математика, поэтому вещественные числа там всегда неточные.



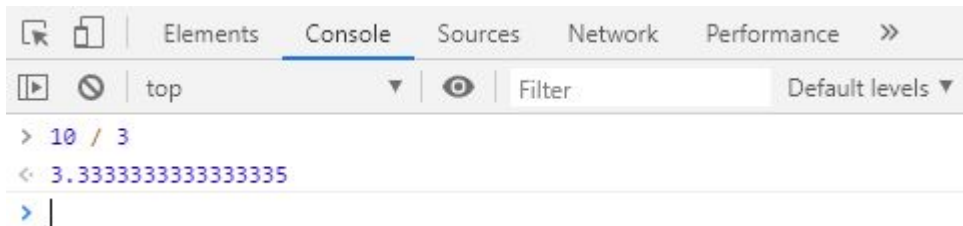
Операторы

Этих операторов вам будет достаточно, для того, чтобы выполнить ДЗ, остальные мы будем изучать при рассмотрении соответствующих примеров.



Консоль

Очень удобно небольшие "кусочки" JS проверять в консоли (просто вводите туда выражение и нажимаете на Enter):



Операторы

Почему мы явно говорили, что с числами можно использовать арифметические операторы? Потому что есть другие типы данных, определяющие свои наборы операторов.



Типы данных



Типы данных

Все типы данных делят на две большие категории:

1. Прimitives:

- a. **Boolean** – true/false
- b. **Null** – null (отсутствие значения)
- c. **Undefined** – undefined
- d. **Number** – число
- e. **String** – строка
- f. **Symbol** – символы
- g. **BigInt** - большие целые числа

2. Объекты:

- a. **Object** – объект



Типы данных

Тип данных определяет то, какие операции можно с этими данными. Остальные типы данных (не числа), мы будем рассматривать следующих лекциях.

Сейчас же давайте посмотрим, что будет, если попробовать в одной операции "смешать" разные типы данных. Посмотрим это на примере самой распространённой связки: числа и строки.



Строка

Строка – это набор символов. Определяется в виде символов, заключённые в одинарные или двойные кавычки* (принято использовать одинарные):

```
1  const input = '10000'; ← Теперь это строка, а не число
2  const exchangeRate = 0.147;
3  const commission = 0.01;
4
5  const amount = input / exchangeRate;
6  const payment = input * (1 + commission);
7
8  console.log(amount);
9  console.log(payment);
```

Но результат в консоли не измениться. Почему? Как можно строку разделить на число?

Примечание*: также могут быть и бэктики – ```, про них будет отдельный разговор.



Приведение типов

JS всячески старается помочь нам и когда встречает данные разных типов, пытается "привести" их к одному типу. Что значит "привести"? Для всех* арифметических операторов он пытается из всех типов данных сделать число.

Например, для строки '10000' выполняется преобразование в число 10000, там, где встречается арифметический оператор:

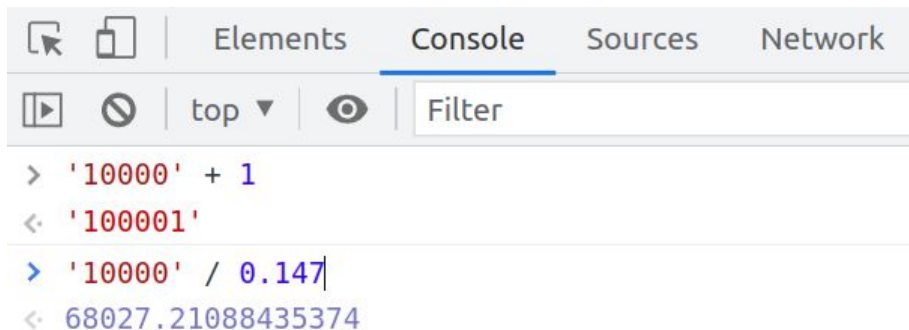
```
const amount = input / 0.147;  
const payment = input * (1 + 0.01);
```

Примечание*: оператор + со строками превращает всё в строки и склеивает их.



Консоль

Мы можем экспериментировать в консоли, вводя туда выражения, которые браузер сразу будет вычислять:



Это достаточно удобно, когда нужно что-то быстро проверить, но и тут будут свои хитрости (увидим их, когда будем обсуждать строгий режим и модули).



Приведение типов

Но что если строку нельзя привести к числу, допустим, там написано слово 'много' (мы с вами скоро будем говорить о том, что пользователь может при желании ввести что угодно и вы никак не сможете этому воспрепятствовать):

```
> '10000' / 0.147  
< 68027.21088435374  
  
> 'много' / 0.147  
< NaN  
  
>
```

Что такое **NaN**? **NaN** – это специальное значение, которое представляет из себя "Не Число" (Not a Number).

В JS достаточно много подобных тонкостей, но на начальном этапе изучения языка лучше запомнить универсальное правило: старайтесь не использовать в одном выражении разные типы данных. Как это сделать, мы с вами посмотрим, когда будем работать с формами.



Приведение типов

Ну и напоследок, ещё раз напомним про тот самый замечательный оператор `+`, который при работе со строками приводит всё к строке:

```
> '10000' + 1  
< "100001"  
  
>
```

Правило достаточно простое: если оператор `+` при работе с двумя операндами встречает хотя бы одну строку, то он всё переделывает в строку.

Важно это запомнить: это одна из топ ошибок при работе с вычислениями в JS.



O JS

Иногда говорят, что в JS слишком уж много подобных "тонкостей" и т.д. На самом деле, смеем вас заверить, что их много в любом языке программирования.

Для нас ключевое: JS – это инструмент и нужно уметь пользоваться этим инструментом (можно долго рассуждать, сколько есть "тонкостей" использования молотка, но когда вам нужно забить гвоздь, вы берёте молоток и забиваете им гвоздь).



ИТОГИ



ИТОГИ

В этой лекции мы обсудили достаточно много важных моментов:

1. Работу с дебаггером (обязательно практикуйтесь в его использовании)
2. Работу с консолью
3. Числа и арифметические операторы
4. Немного поговорили о других типах данных и о приведении типов

Это те основы, на которых мы будем строить решение дальнейших задач и, уже начиная со следующей лекции, начнём рассматривать некоторые тонкости языка (`var`, `"strict mode"`, `TDZ` и т.д.)



ДОМАШНЕЕ ЗАДАНИЕ



Орг.моменты

Практикум состоит из 8 обязательных занятий. Мы выкладываем новые занятия каждый понедельник в 14:00 (по Душанбе), кроме первой недели.

Каждое воскресенье в 23:59 (по Душанбе) дедлайн сдачи домашнего задания. Дедлайн – это предельный срок, до которого вы должны сдать ДЗ.

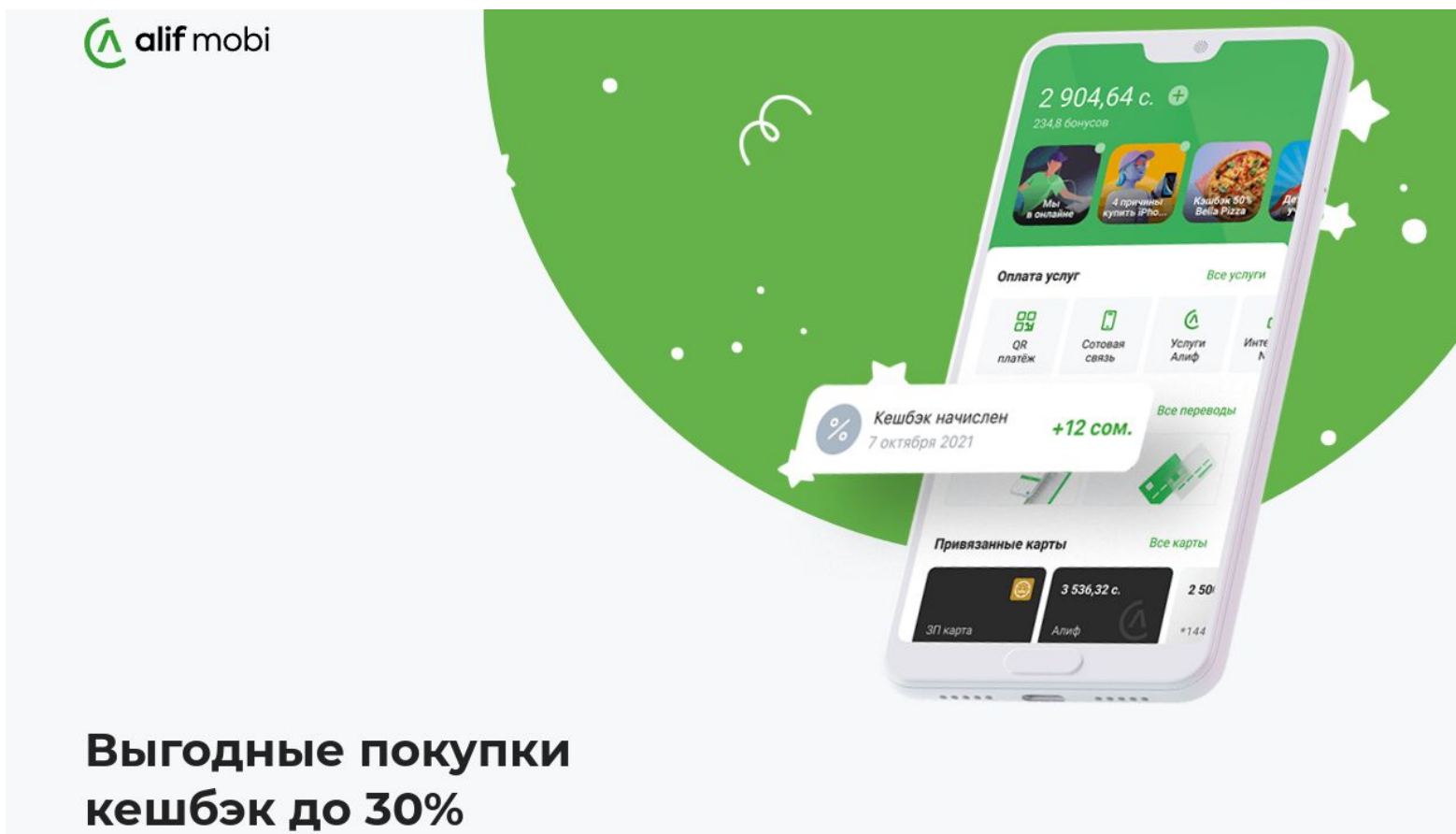
Если не успеете сдать в срок домашнее задание, тогда этот практикум будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.

Все вопросы вы сможете задавать в [Телеграм канале](#).



ДЗ №1: Кэшбэк

Давайте посмотрим на сервис <https://cashback.alif.tj>:



ДЗ №1: Кэшбэк

Правила сервиса гарантируют вам с любой покупки получение кэшбэка в зависимости от магазина.

Вам необходимо написать программу, которая исходя из суммы покупки и кэшбэка конкретного магазина высчитывает итоговый кэшбэк. Важно: округлять до сомони не нужно (мы разберём вопросы округления позже).



ДЗ №1: Кэшбэк

Чтобы автоматизированная система смогла проверить вашу задачу, необходимо выполнить ряд требований:

1. Переменная для вводимой суммы покупок должна объявляться, как `const input = число;` (например, `const input = 1000`), переменная для размера % кэшбэка магазина как `const storePercent = число;` (например, `const storePercent = 30;` для 30% кэшбэка)
2. Переменная для результатов должна называться `cashback`

```
1  const input = 1000;  
2  const storePercent = 30;  
3  
4  const cashback = ...;  
5  
6  console.log(cashback);
```

первоначальные значения должны быть такие!



ДЗ №1: Кэшбэк

Важно: бот сначала запустит программу с вашими данными, а затем заменит значения некоторых переменных и перезапустит приложение.

Проект должен храниться в каталоге **cashback** (т.е. внутри вашего архива должен быть каталог **cashback**).

Бот будет проверять:


1. HTML-файл (наличие подключения в нём JS)
2. JS-файл
3. Вывод в консоль



ДЗ №2: Депозитная карта

Представим, что банк предоставляет следующий продукт:

Получайте доход каждый месяц
на депозитную карту

Сумма вклада	Ожидаемый годовой доход
100000 СОМОНИ	В СОМОНИ от 4000 до 6000
	Ожидаемый ежемесячный доход
Пример расчета процентов по вкладу носит исключительно информационный характер и не является публичной офертой.	В СОМОНИ от 333 до 500



ДЗ №2: Депозитная карта

Что нужно сделать: нужно написать небольшую программу, которая считает четыре значения (см. предыдущий слайд):

1. Ежемесячный доход от
2. Ежемесячный доход до
3. Ежегодный доход от
4. Ежегодный доход до



ДЗ №2: Депозитная карта

Чтобы автоматизированная система смогла проверить вашу задачу, необходимо выполнить ряд требований:

1. В архиве должен быть каталог `deposit`
2. Переменная для вводимой суммы вклада должна объявляться как:
`const input = число;` (например, `const input = 10000`)
3. Переменные для результатов должны называться соответственно:
 - 3.1. `maxMonthly`
 - 3.2. `minMonthly`
 - 3.3. `maxYearly`
 - 3.4. `minYearly`

Важно: округлять до сомони не нужно (мы разберём вопросы округления позже).



ДЗ №2: Депозитная карта

```
1  const input = 1000;
2
3  // ваш код
4
5  const maxMonthly = ...;
6  const minMonthly = ...;
7  const maxYearly = ...;
8  const minYearly = ...;
9
10 console.log(maxMonthly);
11 console.log(minMonthly);
12 console.log(maxYearly);
13 console.log(minYearly);
```

← вместо ... нужно подставить
ваши выражения



ДЗ №2: Депозитная карта

Важно: бот сначала запустит программу с вашими данными, а затем заменит значения некоторых переменных и перезапустит приложение.

Проект должен храниться в каталоге `deposit` (т.е. внутри вашего архива должен быть каталог `deposit`).

Бот будет проверять:

1. HTML-файл (наличие подключения в нём JS)
2. JS-файл
3. Вывод в консоль



Спасибо за внимание

alif skills

2023г.

